Università degli Studi di Parma
Dipartimento di Ingegneria e Architettura
Sistemi operativi e in tempo reale – a.a. 2022/23

Costrutti linguistici per la programmazione concorrente in ambiente globale *Monitor*

prof. Stefano Caselli
stefano.caselli@unipr.it

Alla ricerca di un tradeoff

- Transizione dall'uso dei semafori (basso livello) alle Regioni Critiche Condizionali, un meccanismo di alto livello supportato dal linguaggio di programmazione
- Vantaggi: elevato livello di astrazione della RCC
- Problemi: overhead! Un filosofo acquisisce o rilascia i chopstick, e filosofi molto lontani da lui vengono inutilmente risvegliati!
- Obiettivo: un meccanismo di sincronizzazione che consenta di controllare chi risvegliare, con un compromesso più equilibrato o comunque regolabile dal programmatore

- Problemi irrisolti nei meccanismi di sincronizzazione visti:
  - I processi *accedono direttamente alle variabili comuni* (ad es., entro le regioni critiche), l'accesso alle risorse del sistema *non é controllato*
  - I problemi di sincronizzazione più complessi risultano ancora *difficili* da esprimere e da comprendere
- Dove è la politica di accesso alla risorsa?
  - ogni processo contiene al suo interno una parte della politica con cui viene gestita la risorsa
  - se la risorsa è condivisa, la politica effettiva risulta dall'*interazione* complessiva delle diverse *politiche locali*
  - ogni processo deve disporre della *conoscenza dettagliata* relativa alle risorse al fine di realizzare la propria politica

Evoluzione dei linguaggi di programmazione

- Lo sviluppo degli strumenti per la programmazione concorrente è influenzato dalla profonda evoluzione dei linguaggi che avviene negli stessi anni
- Necessità della programmazione sia sequenziale che concorrente di gestire la complessità: incapsulamento, information hiding, programming in the large
- Anche con le RCC «ogni processo deve disporre della *conoscenza dettagliata* relativa alle risorse al fine di realizzare la propria politica»
- → Primi sviluppi della programmazione object-oriented con il concetto di *tipo di dato astratto* (ADT, Abstract Data Type)

Tipo di dato

- Il tipo di dato identifica:
  - insieme dei *valori* che un dato di quel tipo può assumere
  - insieme delle *operazioni* che possono agire su dati di quel tipo (tutte e sole le operazioni significative per dati di quel tipo)
- Nei linguaggi di programmazione con controllo forte dei tipi:
  - tipi elementari (predefiniti dal linguaggio)
  - tipi astratti (definiti dal programmatore a partire da quelli elementari)

Tipo di dato

- Problemi insiti nella realizzazione debole del tipo astratto:
  - non viene imposto al programmatore di specificare, oltre ai valori associati al nuovo tipo, anche le *operazioni* per manipolare oggetti del tipo
  - non è fornito un meccanismo di *incapsulamento*; la rappresentazione di un tipo astratto è visibile ovunque sia visibile il nuovo tipo (eventuali modifiche alla realizzazione interna del tipo si propagano)

Tipo di dato astratto (loose syntax) (Dahl, 1970)

```
type <nome del tipo> = class {
  <dichiarazione di variabili locali>
procedure entry <nome1>( ... );
  begin  ...   end;
procedure entry <nome2>( ... );
  begin  ...   end;
procedure <nome3> ( ... );
  begin  ...   end;
procedure <nome4> ( ... );
  begin  ...   end;

begin <statement iniziale> end;
}
```

Keyword *class*: identifica definizione ADT

Struttura dati che rappresenta la realizzazione della astrazione

Keyword *entry*: procedure e/o funzioni esportate (note) al di fuori del tipo
Sono le sole operazioni per agire su istanze (oggetti) del tipo

Manca keyword *entry*: procedure e/o funzioni locali al tipo. Possono essere richiamate solo dalle procedure entry; non note all'esterno

La definizione del tipo include sempre una *inizializzazione* valida per tutte le istanze

Fine della definizione del tipo

Tipi di dato astratti

- Scope rules enfatizzano incapsulamento e hiding
- All'esterno della definizione di tipo di dato astratto sono visibili *solo i nomi delle procedure entry*
- Normalmente le procedure di un tipo di dato astratto, entry o locali, possono operare soltanto sui propri parametri, sugli identificatori locali e sulle variabili locali al tipo di dato astratto: *niente è noto dall'esterno*
- I dati locali all'istanza del tipo di dato astratto *sopravvivono* indipendentemente dalle chiamate alle procedure del tipo → i dati locali definiscono uno *stato*

Tipo di dato astratto: esempio

```
type  T-coda = class {
 var     buffer: array [0 .. N-1] of T;
         testa, coda: 0 .. N-1;
         cont: 0 .. N;
 procedure entry inserzione(x: T);
   begin
      if  cont = N then <overflow>;
      else begin
             buffer [coda] := x;
             coda := (coda + 1) mod N;
             cont := cont + 1
            end
   end
procedure entry estrazione(var x: T);
   begin
      if  cont = 0 then <underflow>;
      else begin
             x := buffer [testa];
             testa := (testa+1) mod N;
             cont := cont - 1
            end
       end
begin
     testa := 0; coda := 0; cont := 0
end
}
```

Tipo di dato astratto

Dichiarazione: var <nome istanza>: <nome del tipo astratto>;

Operazione: <nome istanza>.<nome proc entry>(parametri attuali);

Esempio:

```
var     a, b, c: T;
        coda1, coda2: T-coda;
begin
        a:=7;
        coda1.inserzione(a);
        coda2.estrazione(b);
        print(b);                       // b?
        ...
end
```

Oggetti astratti

- In molte applicazioni la *complessità di alcune strutture dati* (oggetti dati) contribuisce in modo determinante alla complessità del problema
- Il ricorso alla *astrazione* ha come scopo di consentire *l'utilizzo* degli oggetti senza conoscere i dettagli dell'*implementazione* (*rappresentazione in memoria*)
- Il comportamento degli oggetti viene descritto tramite un insieme di operazioni che sono associate agli oggetti e che costituiscono *l'unico strumento* per operare sull'oggetto

Oggetti astratti

- Il programmatore può ignorare il *tipo di realizzazione interna (implementation)* adottata per l'oggetto
- Le *operazioni* utilizzate dal programmatore costituiscono a loro volta delle *astrazioni funzionali*
- Per realizzare il concetto di *oggetti astratti* occorre:
  - *definire* tutte le operazioni applicabili agli oggetti di un tipo
  - *nascondere* la rappresentazione degli oggetti all'utente
  - *garantire* che l'utente possa manipolare gli oggetti solo tramite le operazioni

Oggetti astratti

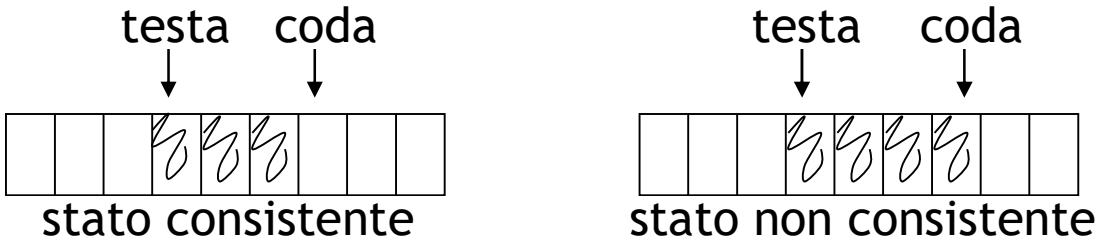
- Un oggetto astratto viene definito come una collezione di oggetti di tipi primitivi e/o di tipi astratti precedentemente definiti
- Un oggetto astratto è caratterizzato dal proprio *stato interno*, costituito dall'aggregazione dei diversi stati (esterni o interni) degli oggetti componenti
- Lo stato interno, non essendo l'oggetto primitivo, in un istante arbitrario può essere consistente o *inconsistente*

Oggetti astratti

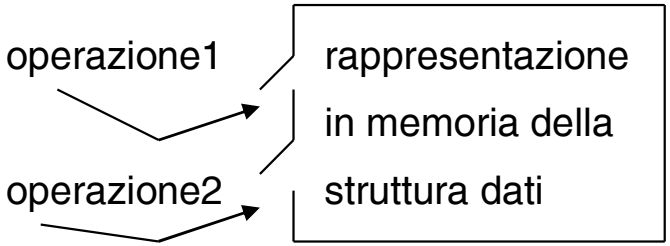
- Un oggetto astratto è caratterizzato anche da uno *stato esterno*, reso visibile mediante *operazioni che comunicano lo stato*. Lo stato esterno è costituito dalla aggregazione di un insieme di stati interni. Ad esempio, per una coda: piena, vuota, né piena né vuota
- Gli stati interni inconsistenti non corrispondono ad alcun stato esterno, cioè vengono nascosti (incapsulamento)
- In presenza di oggetti astratti *condivisi*, occorre garantire che le operazioni su di essi lascino l'oggetto in uno stato interno consistente
- La distinzione tra *tipo di dato astratto* ed *oggetto astratto* è solo la maggior enfasi sugli aspetti di aggregazione e di stato

Proprietà degli oggetti astratti

- Garanzia di *consistenza* di una struttura dati:



- *Protezione*:



information hiding

- *Modificabilità dei programmi*:
  - Se viene modificata la rappresentazione in memoria occorre modificare *solo le operazioni*

Monitor

```
type <nome del tipo> = monitor {
  <dichiarazione di variabili locali>
procedure entry <nome1>( ... );
  begin  ...   end;
procedure entry <nome2>( ... );
  begin  ...   end;
procedure <nome3> ( ... );
  begin  ...   end;
procedure <nome4> ( ... );
  begin  ...   end;

begin <statement iniziale> end;
}
```

Keyword *monitor*: identifica definizione di un ADT per sincronizzazione

E' un *oggetto astratto condiviso* definito da struttura dati, procedure entry e non entry, inizializzazione dello stato come nell'ADT

A questi elementi si aggiunge la specifica della sincronizzazione

Le istanze di un monitor sono accessibili contemporaneamente da più thread!! → problemi di consistenza e ordinamento degli eventi di thread diversi

Monitor

- Le istanze di un monitor sono accessibili contemporaneamente da più processi
- → per mantenere la consistenza della struttura dati del monitor ed evitare interferenze, le *procedure entry* devono essere eseguite in *mutua esclusione*
- Nei linguaggi che offrono nativamente il costrutto Monitor tale *regola di sincronizzazione* è garantita dal *compilatore*
- Se la sincronizzazione è realizzata mediante API di libreria (es. POSIX), è il programmatore che inserisce i *mutex* seguendo il *monitor come pattern*

Monitor

- Lo scopo del monitor è *controllare l'assegnazione* di una risorsa a thread concorrenti in base a determinate politiche di gestione
- L'assegnazione avviene secondo *due livelli*:

1. *Garanzia che un solo thread alla volta abbia accesso al monitor*

   → Le procedure del monitor sono eseguite da un thread alla volta

2. *Controllo dell'ordine con cui i thread hanno accesso alla risorsa*

   In funzione dello stato della risorsa, la procedura può sospendere il thread in una coda locale al monitor; la sospensione comporta la *liberazione del monitor*

Monitor

- Controllo dell'assegnazione della risorsa protetta dal Monitor:

- Per quanto riguarda il 1° livello, la soluzione è la presenza di un *unico* semaforo mutex (v.i. = 1) per proteggere *ogni procedura entry* del monitor. L'accesso al monitor è *serializzato*
- Per quanto riguarda il 2° livello, si introducono *nuove variabili* denominate *variabili di tipo condizione*

Variabili di tipo condizione

- Ogni variabile di tipo condizione rappresenta *una coda* nella quale sono sospesi i thread
- Una variabile condizione viene realizzata mediante un campo valore ed un campo puntatore (alla coda)
- Il programmatore può prevedere *tante variabili di tipo condizione* quante sono le *condizioni logiche distinte* per cui un thread può essere ritardato
- Le condizioni logiche che interessa distinguere possono essere *più o meno specifiche*. Nel caso limite, ci può essere una condizione per ogni thread: in tal caso le variabili di tipo condizione hanno una funzione analoga ai *semafori privati*

Variabili di tipo condizione

- Le procedure del monitor agiscono su tali variabili tramite le operazioni **cond.wait** e **cond.signal**
- (denominazione alternativa: *cond.delay* e *cond.continue*)

- L'esecuzione della operazione **cond.wait** *sospende sempre il thread* e lo inserisce nella coda associata alla variabile *cond*. La sospensione libera il monitor.
- L'esecuzione della operazione **cond.signal** *rende attivo un thread* in attesa nella coda individuata dalla variabile *cond*. Nel caso di coda vuota *non ci sono effetti collaterali*.

Esempio: Produttori - Consumatori

```
type  mailbox = monitor {
        var      buffer: array [0 .. N-1] of  messaggio;    /* rappr.  oggetto  */
                 testa, coda: 0 .. N-1;
                 cont: 0 .. N;
                 non_pieno, non_vuoto: condition;
 procedure entry  send (x: messaggio);
        begin
                 if  cont = N  then  non_pieno.wait;
                 buffer [coda] := x;
                 coda := (coda + 1) mod  N;
                 cont := cont + 1;
                 non_vuoto.signal
        end
```

Esempio: Produttori - Consumatori

```
 procedure entry  receive (var  x: messaggio);
        begin
                if  cont = 0 then non_vuoto.wait;
                x := buffer [testa];
                testa := (testa + 1) mod  N
                cont := cont - 1;
                non_pieno.signal
        end
 begin  cont := 0; testa := 0; coda := 0  end
}               /* fine definizione del tipo */
```

Dichiarazioni delle variabili:

```
var  a, b:   messaggio;
var  B, D:   mailbox;
       //il tipo del Monitor
```

Uso della istanza di Monitor da parte dei thread T1 e T2:

```
T1:   B.send(a);
T2:   B.receive(b);
```

Esempio: Produttori - Consumatori

- Soluzione del problema Produttori-Consumatori mediante un'istanza del monitor mailbox:

var a, b: messaggio;
B: mailbox;

Pi	Ci
...	...
B.send(a);	B.receive(b);
...	...

- La sincronizzazione dei thread non scompare (come con la RCC) ma è incapsulata nella definizione del Monitor

Monitor per Produttori-Consumatori

- Nell'ambito dei thread Pi e Ci scompare qualunque riferimento alle primitive di sincronizzazione. Le primitive sono confinate e mascherate dalle procedure send e receive, scritte *una volta per tutte* all'interno del monitor

- Rispetto alla soluzione mediante i semafori, qui non è più possibile inserire e prelevare *in parallelo* su posizioni diverse: il compilatore introduce *un unico semaforo mutex* per tutte le procedure del monitor, impedendo accessi contemporanei sull'intera risorsa

Monitor Produttori-Consumatori

- Esercizio
- Cosa accade con:

```
var     a, b: messaggio;
        B, D: mailbox;
```

P1	P2
...	...
B.send(a);	D.receive(b);

Esempio: monitor come allocatore di risorse

```
type  allocatore = monitor {
  var     occupato: boolean;
          libero: condition;
  procedure entry richiesta {
                    if occupato then libero.wait;
                    occupato := true;
                    }
  procedure entry rilascio {
                    occupato := false;
                    libero.signal;
                    }
  begin   occupato := false;  end
}
```

Uso del monitor

```
// creazione istanza:

var alloc: allocatore;

// uso istanza:

    alloc.richiesta;
    <uso della risorsa>;
    alloc.rilascio;
```

Realizzazione del monitor tramite semafori

- Il *compilatore* associa *ad ogni istanza* di monitor:
  - un semaforo **mutex** inizializzato ad 1 per la *mutua esclusione* delle procedure del monitor
  - un semaforo **urgent** inizializzato a 0 per effettuare la *preemption* dei thread segnalanti
  - un contatore **urgentcount** inizializzato a 0 per conteggiare in ogni istante i thread segnalanti sospesi
  - per ogni variabile *cond* di tipo *condition* un semaforo **condsem** inizializzato a 0 ed un contatore **condcount** inizializzato a 0, per realizzare cond.wait e cond.signal

Realizzazione del monitor tramite semafori

- Il *compilatore* espande ogni *procedure entry* con un **prologo** ed un **epilogo**:

  - prologo: *wait* (mutex);
  - epilogo: *if* urgentcount > 0
    then signal (urgent);
    else signal (mutex);

- L'epilogo garantisce che il monitor sia libero in assenza di thread al suo interno

Realizzazione del monitor tramite semafori

- Il *compilatore* traduce le operazioni *cond.wait* e *cond.signal*
- Variabili di sistema per la realizzazione delle variabili condizione:
  condsem: *semaforo* associato a ciascuna variabile condizione (v.i.=0)
  condcount: *intero* associato a ciascuna variabile condizione (v.i.=0)

cond.wait:

```
condcount := condcount + 1;          /* tiene traccia della sospens. */
if urgentcount > 0
then signal(urgent);                 /* riattiva un thread preempted */
else signal (mutex);                 /* o libera del monitor */
wait (condsem);                      /* sospens. su semaforo condsem */
condcount := condcount - 1;
```

Realizzazione del monitor tramite semafori

```
        cond.signal:
 urgentcount := urgentcount + 1;
 if condcount > 0 then
 begin                                   /* se ci sono thread sospesi */
        signal(condsem);                 /* risveglia */
        wait(urgent);                    /* sospensione thread segnalante */
 end
urgentcount := urgentcount - 1;
```

Realizzazione del monitor tramite semafori

- Scopo del semaforo urgent *(semantica Hoare)*:
- Occorre garantire ai thread sospesi (cond.wait), non appena riattivati da una cond.signal, la *immediata ripresa* della procedura interrotta
- Infatti occorre evitare che il thread segnalante, mantenendo il controllo della CPU, *modifichi successivamente la condizione cond* prima che il thread segnalato sia andato in esecuzione (il thread segnalato andrebbe comunque in esecuzione)
- → Viene introdotto un semaforo *urgent* (v.i. = 0). Il thread che esegue *cond.signal* si sospende tramite *wait(urgent)*, si ha cioè una *preemption del segnalante*

Realizzazione del monitor tramite semafori

- Scopo del semaforo urgent *(semantica Hoare)*:
- Prima di abbandonare il monitor con signal(mutex) occorre verificare *che non ci siano thread sospesi* su urgent: questi thread devono avere priorità su eventuali richieste dall'esterno

- L'uscita da una procedura del monitor è quindi:

  if urgentcount > 0 *then signal*(urgent)
  else signal(mutex)

- Semantica denominata anche '*Signal and Wait*'

Realizzazione del monitor - varianti

- La gestione del monitor è piuttosto onerosa.
- Soluzione Concurrent Pascal (Brinch-Hansen) '*Signal and Exit*'
- Spesso è possibile scrivere procedure del monitor in cui la *cond.signal* compare come *ultima operazione della procedura*, e pertanto non c'è più pericolo di modifica delle variabili condizione. In tal caso è possibile semplificare la realizzazione mediante semafori.
- Non c'è più la necessità del semaforo *urgent* e del contatore *urgentcount*, per cui:

  prologo: *wait*(mutex);

  epilogo: *signal*(mutex); /* se la procedura non contiene cond.signal */

Realizzazione del monitor - varianti

```
cond.wait:      condcount := condcount + 1;
                signal(mutex);
                wait(condsem);
                condcount := condcount - 1;

cond.signal:    if  condcount > 0
                then signal(condsem);
                else signal(mutex);     /* solo in questo caso
                                           il monitor viene liberato */
```

- Ove presente, la cond.signal *rappresenta anche l'epilogo* della procedura

Realizzazione del monitor - varianti

- La procedure del monitor consentono di definire *una qualunque politica di assegnazione della risorsa*. Ciò non vale per la *politica di risveglio* dei thread, che si basa su una *signal* del SO o della libreria con meccanismi impliciti (FIFO)
- Una proposta: consentire una politica basata su priorità indicando la *priorità* in fase di sospensione:

  cond.wait(priorità)

- Il SO provvederà a risvegliare il thread a priorità più elevata
  - Quesito: dove potrebbe servire?

Realizzazione del monitor - varianti

- Variabili condizione come *code monothread* (una condizione per ogni thread, come semafori privati): il programmatore ha il *controllo completo* sulla scelta del thread da risvegliare
  - Occorre conoscere *a priori* quanti thread agiscono sulla risorsa per prevedere nella definizione del tipo le relative variabili condizione
  - Ogni thread quando esegue una procedure del monitor specifica la propria identità con un indice
- Altre *funzioni primitive* su variabili di tipo condizione:
  cond.queue
  - Verifica la presenza nella coda *cond* di almeno un thread sospeso
  - Esempio: *if* cond.queue *then* ...

Esempio: Lettori-Scrittori

```
type  lettori_scrittori = monitor {
 var    num_lettori: integer; occupato: boolean;   /* N. lettori su risorsa */
        ok_lettura, ok_scrittura: condition;        /* e presenza scrittore */
 procedure entry  Inizio_lettura;
  begin
        if  (occupato or ok_scrittura.queue) then  ok_lettura.wait;
        num_lettori := num_lettori + 1;
        ok_lettura.signal;              /* risveglio a catena dei lettori già nel  */
  end                                   /* monitor dopo accesso di scrittore    */
procedure entry  Fine_lettura;
 begin
        num_lettori := num_lettori - 1;
        if  num_lettori = 0 then  ok_scrittura.signal;
 end
```

Esempio: Lettori-Scrittori

```
procedure entry  Inizio_scrittura;
 begin
        if ((num_lettori <> 0) or occupato) then  ok_scrittura.wait;
        occupato := true;
 end
 procedure entry  Fine_scrittura;
 begin
        occupato := false;
        if  ok_lettura.queue then  ok_lettura.signal;
                          else  ok_scrittura.signal;
 end
 begin  num_lettori := 0;    occupato := false;  end
}
```

Soluzione *weak-weak writer preference*!

Sostituendo con:
'if ok_scrittura.queue then ok_scrittura.signal; else ok_lettura.signal;'
si ha *strong writer preference*

Esercizi su Monitor Lettori-Scrittori

- Sostituire l'uso delle primitive *queue* su variabili condizione con variabili intere che conteggiano i thread in attesa
- Sviluppare Monitor per diverse priorità di Lettori-Scrittori

Filosofi a cena: soluzione mediante *monitor*

```
type  dining-philosopher = monitor
 var  state:  array  [0 .. N - 1]  of   (thinking, hungry, eating);
      self:   array  [0 .. N - 1]  of   condition;

 procedure entry pickup (i:  0 .. N-1)  {
                 state [i] := hungry;
                 if  (state [(i+N-1)mod N] <> eating
                     and state [(i+1)mod N] <> eating)
                 then  state [i] := eating;
                 else   self [i].wait;
 }
```

Filosofi a cena: soluzione mediante *monitor*

```
procedure entry putdown (i: 0 .. N-1)  {
        state [i] := thinking;
        if  state [(i+N-1)mod N] = hungry and state [(i+N-2)mod N] <> eating
        then  begin
                state [(i+N-1) mod N] := eating;
                self [(i+N-1) mod N] .signal;
              end
        if  state [(i+1) mod N] = hungry and  state [(i+2) mod N] <> eating
        then  begin
                state [(i+1) mod N] := eating;
                self [(i+1) mod N] .signal;
              end
}
```

Filosofi a cena: soluzione mediante *monitor*

```
type  dining-philosopher = monitor {
  var  state:  array  [0 .. N - 1]  of   (thinking, hungry, eating);
       self:   array  [0 .. N - 1]  of   condition;

  procedure entry  pickup (i:  0 .. N-1);
  procedure entry  putdown (i:  0 .. N-1);

  begin  for  i := 0  to  N-1 do  state [i] := thinking;  end
} //end type
```

- L'inizializzazione è parte della definizione del tipo di dato astratto

Filosofi a cena: soluzione mediante *monitor*

- Creazione dell'istanza:

  var dp: dining-philosopher;

Philosopher_i:

repeat

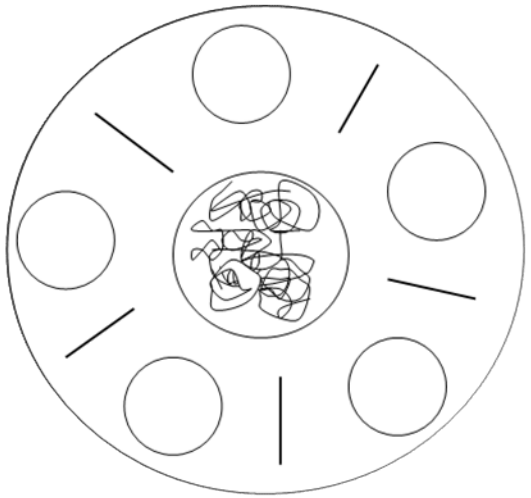
<think>;

dp.pickup(i);

<eat>;

dp.putdown(i);

forever



Filosofi a cena: soluzione mediante *monitor*

- Discussione:
- Soluzione analoga a quella dei semafori privati, con una variabile condizione per ogni filosofo
- E' possibile realizzare una soluzione mediante Monitor che preveda un'unica variabile condizione? Sotto quali ipotesi?

Monitor e deadlock

- *Ricorsività* di una procedura entry:

  Rilevabile in fase di compilazione

- All'interno di un *singolo monitor due thread si bloccano* ognuno aspettando di essere svegliato dall'altro

  Errore limitato al codice del monitor; generalmente facile da localizzare e correggere

- Due procedure `a`, `b`, di due *monitor diversi* `A`, `B`, che *si richiamano reciprocamente*

  Tipica situazione di deadlock dovuta ad una *dipendenza mutua*. Può essere evitata mediante un *ordinamento gerarchico*. In tal caso (evitando monitor mutuamente ricorsivi) può essere rilevata in fase di compilazione

Monitor e deadlock

- Due procedure `a1, b1`, di due monitor diversi `A, B`, con `a1` che richiama `b1` e `b1` che esegue una cond.wait, mentre la cond.signal viene eseguita da una procedura `b2` di `B` che, in tale stato, può essere richiamata solo da una procedura di `A`
- ==> Problema delle *chiamate innestate*

Modifica esempio Lettori-Scrittori

```
type  lettori_scrittori = monitor {
  var    num_lettori:   integer;  occupato:  boolean;
         ok_lettura, ok_scrittura:   condition;
  procedure entry  Inizio_lettura;
   begin
         num_lettori := num_lettori + 1;
         while (occupato or ok_scrittura.queue) then ok_lettura.wait;
         ok_lettura.signal; /* risveglio a catena lettori già nel monitor  */
   end                                          /* dopo accesso di scrittore    */
  procedure entry  Fine_lettura;
   begin
         num_lettori := num_lettori - 1;
         if  num_lettori = 0 then  ok_scrittura.signal;
   end
```

Modifica esempio Lettori-Scrittori

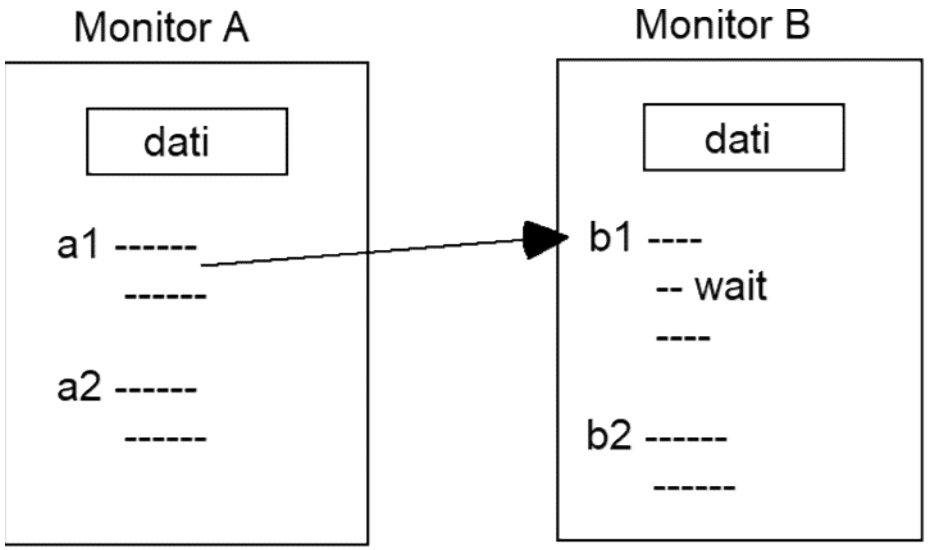
```
procedure entry  Inizio_scrittura;
  begin
        if  ((num_lettori <> 0) or occupato) then ok_scrittura.wait;
        occupato := true;
   end
procedure entry  Fine_scrittura;
  begin
        occupato := false;
        if  ok_lettura.queue then  ok_lettura.signal;
                                else  ok_scrittura.signal;
  end
 begin  num_lettori := 0;   occupato := false;   end
}
```

Discussione

- Sono state operate due modifiche molto contenute alla versione inziale del monitor. Si tratta di modifiche singolarmente innocue. Nessuna delle due da sola determina un malfunzionamento
- C'è tuttavia una possibilità di deadlock interna al monitor
- (Relativamente) facile da rilevare e da correggere grazie alla localizzazione del problema all'interno del monitor (?)

Chiamate innestate a monitor

- Due monitor A e B
- a1 richiama b1 e b1 esegue una cond.wait
- la cond.signal viene eseguita da una procedura b2 di B che, in tale stato, può essere richiamata solo da una procedura di A
- Problema: cond.wait *libera* il monitor B a cui appartiene la procedura b1 che esegue la wait, ma mantiene *occupato* il monitor *chiamante* A

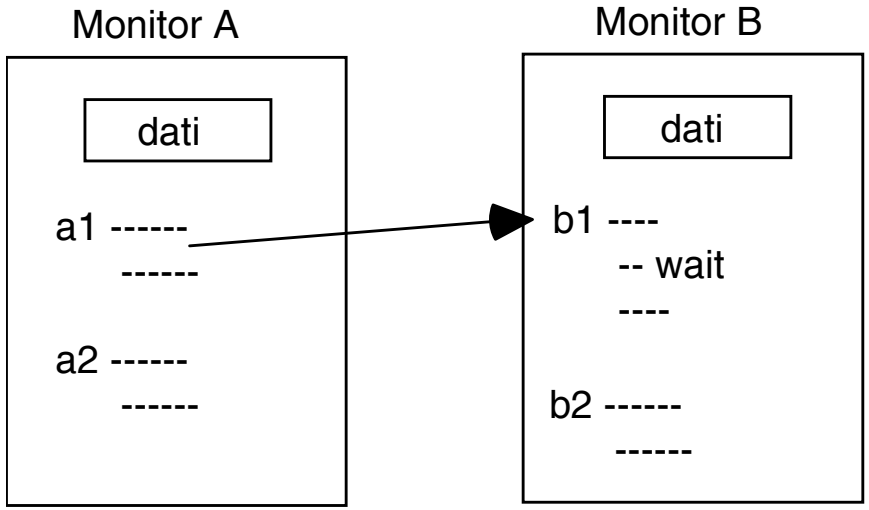


Soluzione imposta da esigenza di congruenza dei dati: non essendo terminata a1, se il monitor A viene liberato un nuovo thread può trovare la struttura dati del monitor in stato non consistente

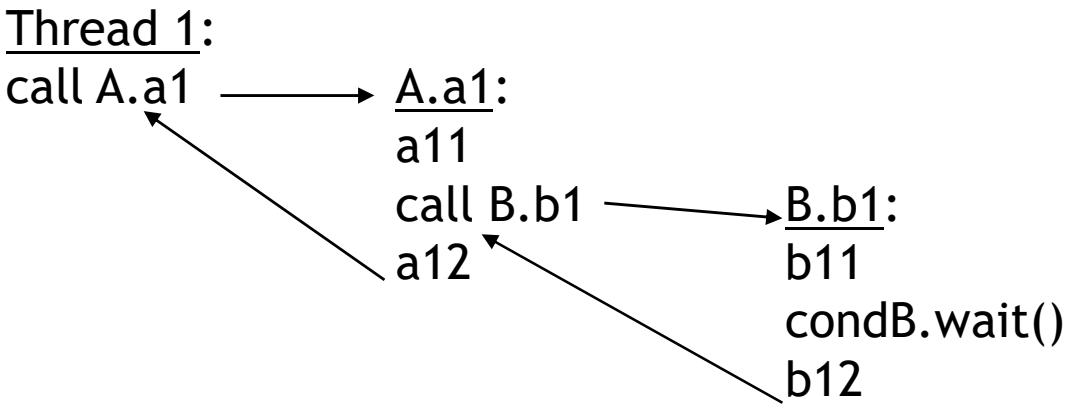
Chiamate innestate a monitor

- Alcuni linguaggi *vietano le chiamate innestate a monitor*, imponendo un'organizzazione più gerarchica ai programmi
- Oppure …
- One big lock vs. several small locks

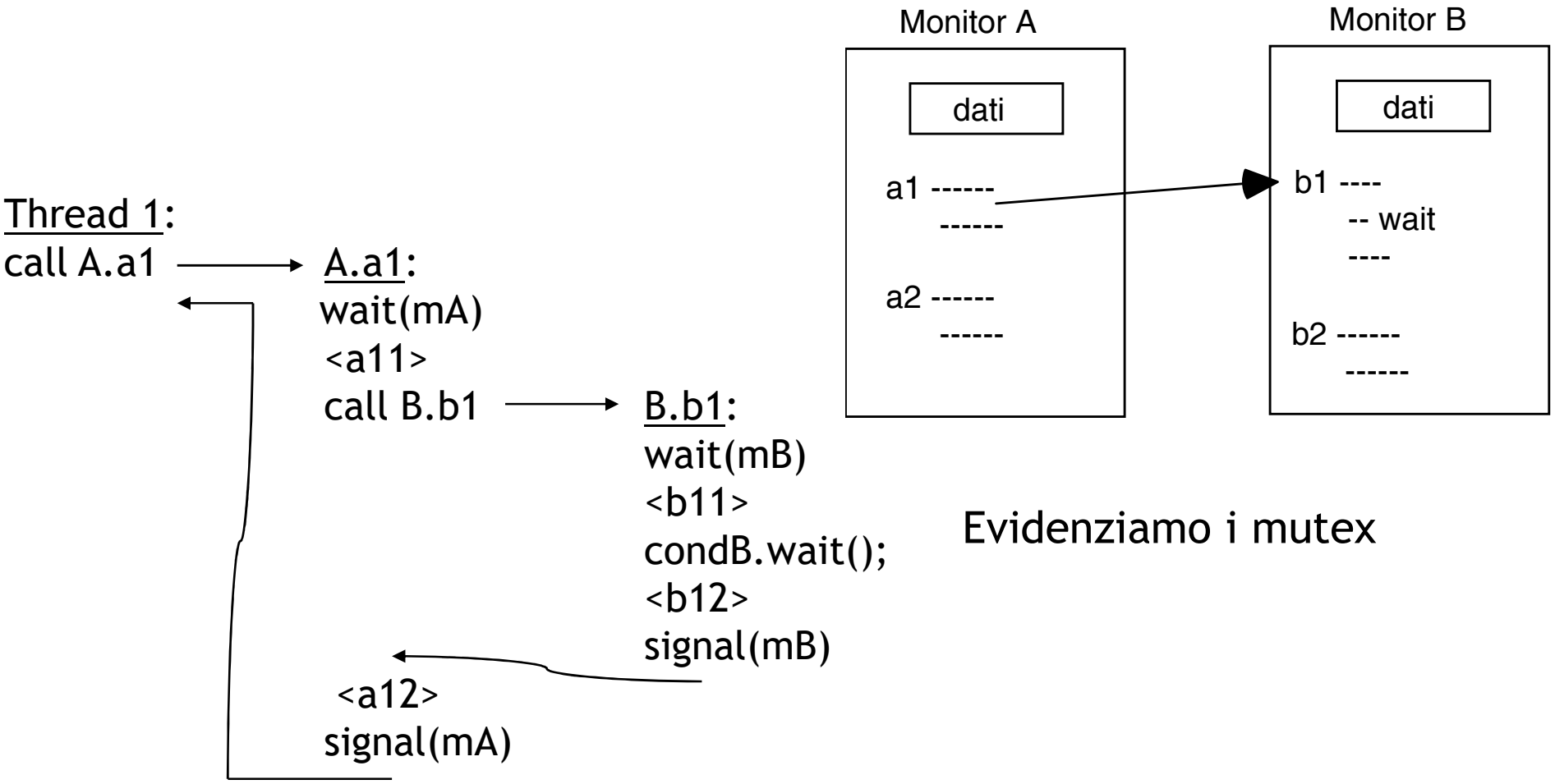
Chiamate innestate a monitor



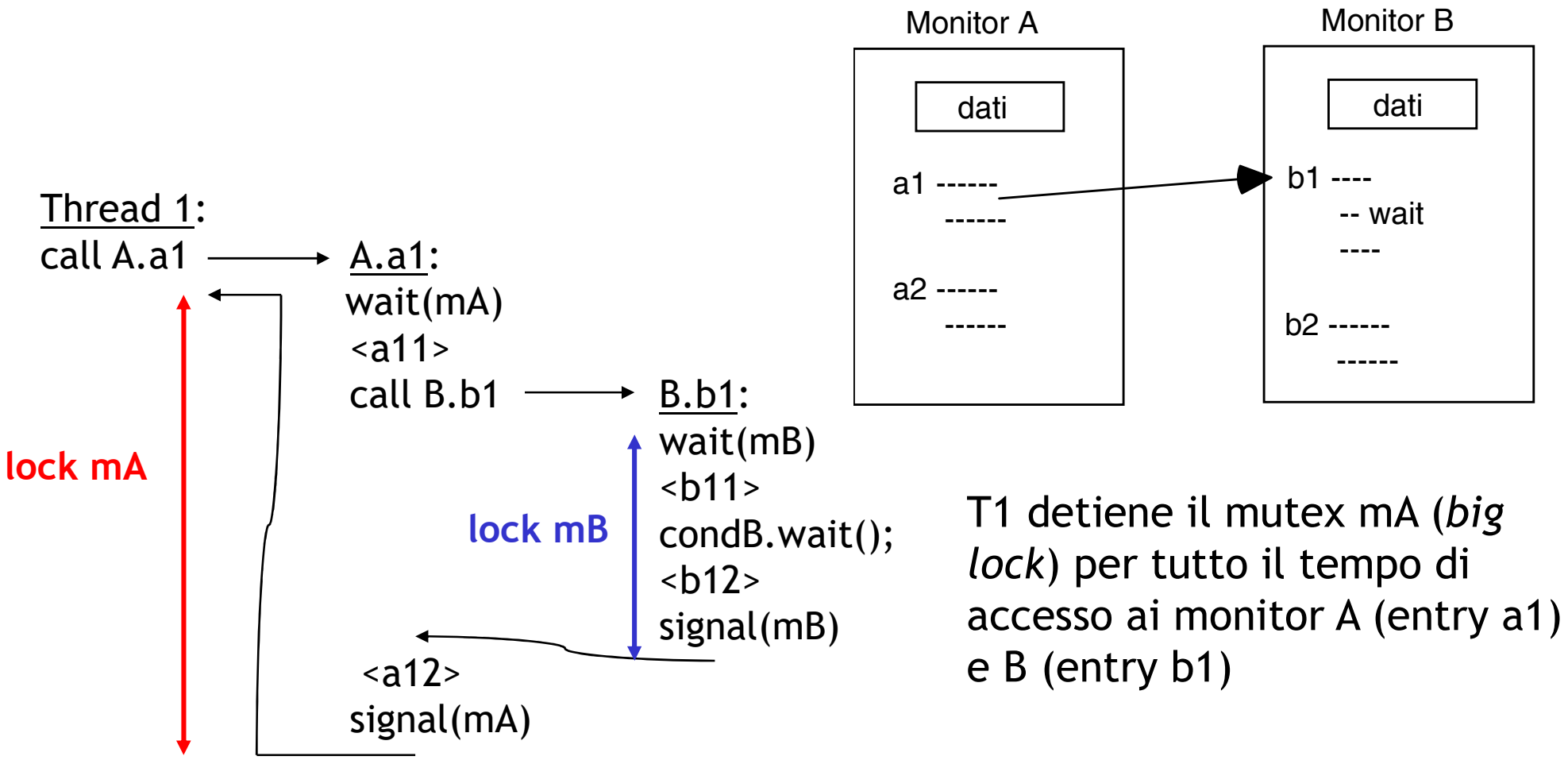
La sequenza di chiamate:



Chiamate innestate a monitor

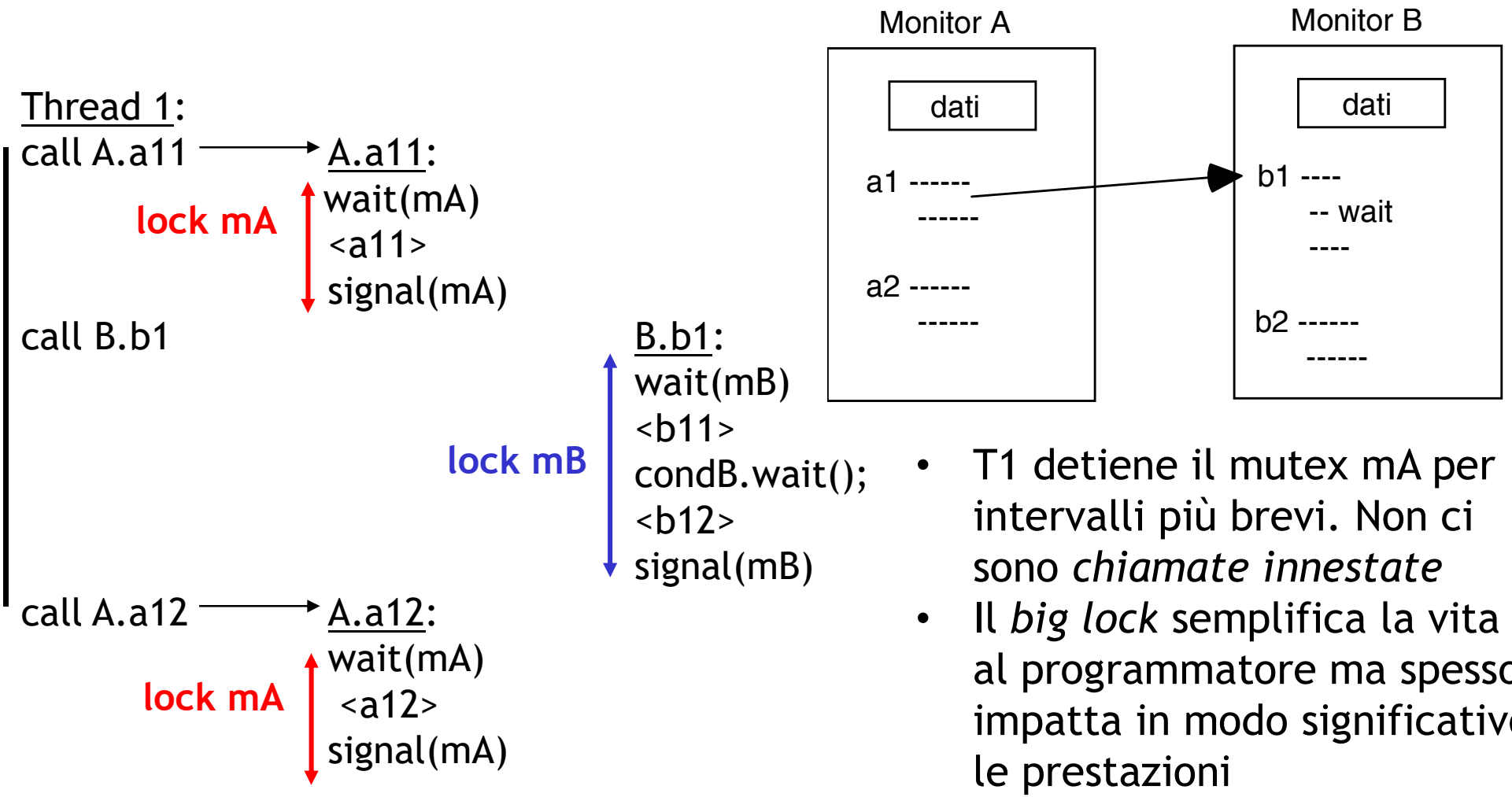


Chiamate innestate a monitor



T1 detiene il mutex mA (*big lock*) per tutto il tempo di accesso ai monitor A (entry a1) e B (entry b1)

Chiamate innestate a monitor



- T1 detiene il mutex mA per intervalli più brevi. Non ci sono *chiamate innestate*
- Il *big lock* semplifica la vita al programmatore ma spesso impatta in modo significativo le prestazioni

Chiamate innestate a monitor

Avoid big lock whenever possibile...
but it ain't free!

Thread 1:
call A.a11 → A.a11:
wait(mA)
<a11>
signal(mA)

lock mA

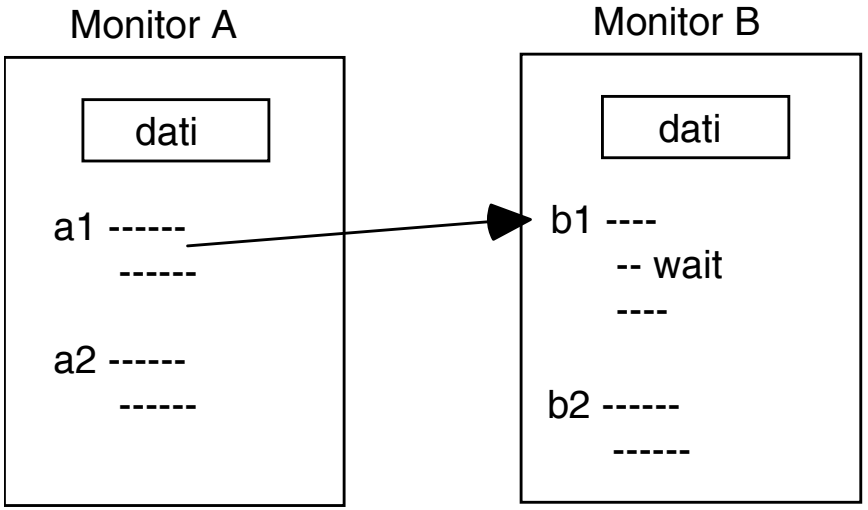
call B.b1

B.b1:
wait(mB)
<b11>
condB.wait();
<b12>
signal(mB)

lock mB

call A.a12 → A.a12:
wait(mA)
<a12>
signal(mA)

lock mA



- T1 detiene il mutex mA per intervalli più brevi. Non ci sono *chiamate innestate*
- Il *big lock* semplifica la vita al programmatore ma spesso impatta in modo significativo le prestazioni

Tipi di monitor: semantiche alternative

- Signal and Wait: Hoare
- Signal and Exit: Brinch-Hansen
- Signal and Continue: Mesa

Tipi di monitor

- Ogni monitor rafforza e mantiene un *invariante* di correttezza dello stato
  - ad esempio: $0 \leq d\text{-}e \leq N$ nel problema produttori-consumatori con buffer limitato
- Ogni volta che il monitor è liberato o commutato l'invariante deve essere valido: un thread che entra nel monitor si aspetta di trovare la risorsa in uno *stato consistente*
- In tutti i tipi di monitor, *l'invariante di base* deve essere ripristinato:
  - prima di eseguire una cond.wait
  - prima di completare la procedura entry ed uscire dal monitor

Tipi di monitor

- Nel monitor *alla Hoare*, una condizione *più specifica* deve essere stata verificata anche prima di una cond.signal
- Ad es., per i produttori-consumatori deve valere: d-e > 0 prima di una cond.signal ad un thread consumatore
- Il thread *segnalato dipende criticamente* da questa condizione logica e non può eseguire se non è soddisfatta
- Soluzione di Hoare:
  - uso del semaforo *urgent* e priorità al segnalato
  - il thread verifica la propria condizione specifica tramite **if**
- E se invece il thread in attesa verificasse la propria condizione sotto un **while** ?

Monitor in semantica Mesa

- E se invece il thread in attesa verificasse la propria condizione sotto un **while** anziché **if** ?
- Con il test della condizione sotto **if** in stile Hoare si ha:
  - valutazione della condizione più semplice (1 sola volta vs. 2 o più con uso del while)
  - più cambi di contesto
  - problema dei risvegli spurii *(spurious wakeup)*
- → Ipotesi: nuova valutazione della condizione dopo il risveglio (quindi uso del **while**) per prevenire risvegli spurii, costa poco!
- Con uso del while, è naturale lasciare *proseguire il thread segnalante* e si riducono anche i context switch
- → un ventaglio di opportunità

Monitor in semantica Mesa

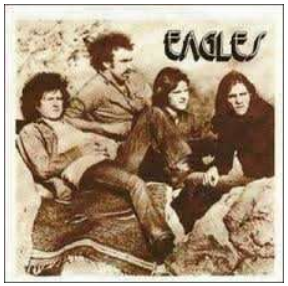
- Linguaggio Mesa (Lampson & Redell, 1980)
- Le primitive (principali) che operano sulle variabili condizione sono denominate *wait* e *notify*
- *Notify* significa che il thread bloccato sulla variabile condizione *potrebbe* poter proseguire! --> *hint*
- L'invariante specifico del thread segnalato non è garantito
- Il thread segnalato *deve comunque valutare nuovamente la condizione*, ed eventualmente ri-sospendersi (stile Mesa):

  while not ok_to_proceed *do* cond.wait *end;*
- anzichè (stile Hoare):

  if not ok_to_proceed *then* cond.wait;

Monitor in semantica Mesa

- Poichè il processo segnalante non subisce preemption:
  - meno context switch
  - non è necessario ristabilire l'*invariante* prima di effettuare *cond.notify* (come accade nel monitor alla Hoare)
  - protezione dal problema degli *spurious wakeup*
  - consente *time-out* sulle attese (cond.timed_wait), *broadcast*, "lassaise faire" programming (hint)



- Il processo segnalato (“notificato”) verrà *prima o poi* posto in esecuzione, in base a decisioni di scheduling globali

Semantica Mesa e semantica Hoare

- Monitor in semantica Mesa significa *monitor in cui il segnalante mantiene il controllo* del mutex del monitor dopo avere segnalato la variabile condizione
- Resta ovviamente possibile ai thread (ma poco prudente) attendere sulle variabili condizione sotto if e non while, o anche in modo incondizionato; in tal caso, il segnalante deve garantire che sia verificato l'invariante specifico del thread
- Anche nei monitor in semantica Hoare è possibile attendere sotto while; la semantica resta Hoare!

Tipi di monitor

- Semantica *signal and exit* → Concurrent Pascal
- Semantica *signal and wait* → Hoare
- Semantica *signal and continue* → Mesa, Java, Pthread

- Da notare che l'adozione di strumenti con semantica *signal and continue* si è consolidata in parallelo con la diffusione delle architetture multiprocessore e multicore

Supporti per la concorrenza in Java

- A ogni oggetto è associata una *variabile di lock*. Per assicurare la *mutua esclusione* si usa la parola chiave `synchronized`:
  - sincronizzazione del metodo:

```
synchronized void f() { /* body */ }
```

  - sincronizzazione dell'oggetto:

```
synchronized (object) {statements}
```

- I metodi dichiarati `synchronized` sono serializzati attraverso la variabile di lock. Il lock viene *acquisito all'accesso del metodo* synchronized e *rilasciato alla sua uscita*, anche se l'uscita avviene per un'eccezione

Supporti per la concorrenza in Java

- Diversamente da quanto accade con un semaforo mutex, il lock è *recursive*, cioè il thread non si blocca se già dispone del lock
- In Java (e con altri linguaggi e librerie, incluso POSIX) *non c'è garanzia di ordinamento temporale* delle richieste di accesso al lock

Supporti per la concorrenza in Java

- Ad ogni oggetto è associata una *variabile condizione* anonima (detta anche *wait set*) manipolabile solo dai metodi `wait, notify, notifyAll`
- I metodi `wait, notify, notifyAll` possono essere invocati solo disponendo già del lock di sincronizzazione associato all'oggetto
- La `wait` rilascia il lock anche se esso è detenuto in modo rientrante. Esiste anche una versione temporizzata (timed wait)

Supporti per la concorrenza in Java

- La `notify` risveglia un thread arbitrario tra quelli in attesa. Il thread che *esegue notify* mantiene il controllo dell'oggetto fino a quando non rilascia il lock
- Il thread che ha *ricevuto il notify* deve riottenere (implicitamente) il lock, e quindi deve sempre attendere che almeno il thread segnalante lo rilasci. Inoltre può essere scavalcato anche da thread esterni => il thread deve valutare nuovamente la condizione
- `notifyAll` effettua il notify su tutti i thread bloccati sulla condizione, che devono tuttavia riguadagnare il lock uno per volta. L'ordine con cui i thread risvegliati verranno eseguiti non è specificato

Supporti per la concorrenza in Java

- Lock e variabili condizione consentono di utilizzare qualsiasi oggetto come monitor!
- Esempio di uso di variabile condizione:

```
public synchronized void entry-method()
                throws InterruptedException {
  while (!cond) wait();
  ++count;
  // qui modifica dei dati del monitor
  notifyAll();
}
```

Supporti per la concorrenza in Java

- Il tipo di monitor che è possibile costruire con i supporti per la concorrenza offerti da Java offre elevata astrazione ma poca selettività, data la presenza di una sola condizione implicita

- Che differenza c'è tra un monitor di questo tipo ed una regione critica condizionale?

Produttore-Consumatore in Java

```
public synchronized void deposit(double value) {
  while (count == numSlots)
      try { wait(); } catch (InterruptedException e) {}
  buffer[putIn] = value;
  putIn = (putIn + 1) % numSlots;
  count++;                          // wake up consumer since
  if (count == 1) notify();         // it might be waiting
}
```

Produttore-Consumatore in Java

```
public synchronized double fetch(){
  double value;
  while (count == 0)
      try { wait();} catch (InterruptedException e) {}
  value = buffer[takeOut];
  takeOut = (takeOut + 1) % numSlots;
  count--;                              //wake up producer since
if (count == numSlots-1) notify(); // it might be waiting
return value;
}
```